Near memory data structure rearrangement

Maya Gokhale
Lawrence Livermore National Laboratory, Livermore, CA
gokhale2@llnl.gov

Scott Lloyd
Lawrence Livermore National Laboratory, Livermore, CA
lloyd23@llnl.gov

Chris Hajas[*]
University of Florida, Gainesville, FL
hajas@hcs.ufl.edu

ABSTRACT

As CPU core counts continue to increase, the gap between compute power and available memory bandwidth has widened. A larger and deeper cache hierarchy benefits locality-friendly computation, but offers limited improvement to irregular, data intensive applications. In this work we explore a novel approach to accelerating these applications through in-memory data restructuring. Unlike other proposed processing-in-memory architectures, the rearrangement hardware performs data reduction, not compute offload. Using a custom FPGA emulator, we quantitatively evaluate performance and energy benefits of near-memory hardware structures that dynamically restructure in-memory data to cache-friendly layout, minimizing wasted memory bandwidth. Our results on representative irregular benchmarks using the Micron Hybrid Memory Cube memory model show speedup, bandwidth savings, and energy reduction. We present an API for the near-memory accelerator and describe the interaction between the CPU and the rearrangement hardware with application examples. The merits of an SRAM vs. a DRAM scratchpad buffer for rearranged data are explored.

CCS Concepts

•Computer systems organization → Parallel architectures; •Hardware → *Buses and high-speed links; Chip-level power issues; Memory and dense storage;*

Keywords

accelerator; data intensive; data rearrangement; energy; memory bandwidth; processing in memory

1. INTRODUCTION

The flattening of processor clock frequency has led to the emergence of many-core CPUs exploiting massive spatial parallelism at a slower clock rate. The challenge of providing data to all the cores is addressed to a certain degree through a deep memory hierarchy with multiple levels of successively larger shared caches and high capacity, high bandwidth stacked DRAM as main memory. The combination of cache hierarchy to exploit spatial/temporal locality and high bandwidth to accommodate streaming access patterns works well for many workloads. However, applications that manipulate complex, linked data structures benefit much less, if at all. These applications often show random rather than streaming access patterns and experience high latency, due both to the random access and to cache pollution when only a small portion of a cache line is used. Unfortunately, the ubiquity of object-oriented, polymorphic design patterns required to solve difficult yet common tasks means that a great many applications have significant phases that fall into this latter data-centric category.

The architecture of stacked 3D memory that uses a base logic layer to manage the DRAM arrays offers an opportunity to accelerate data access for these applications. Since the base layer is separate from the DRAM, additional functionality can be introduced to process data within the memory package, exploiting the large in-package bandwidth and reduced latency, without modifying DRAM fabrication. The emergence of stacked DRAM with a distinct processing layer such as the Micron Hybrid Memory Cube [11] has ignited new enthusiasm from processing-in-memory researchers. For many decades, the quest for memory integrated computing has resulted in novel designs [14, 9, 3, 13, 6, 5, 8], with negligible commercial follow-through due to cost. With 2.5D and 3D stacked DRAM connected to logic with through-silicon vias (TSVs), the barrier to placing compute logic near memory is greatly reduced.

We have designed and emulated a hardware/software system modeled on the HMC that places specialized *data rearrangement hardware* on the base layer. Our approach is a new form of near memory computing that uses hardware in the base logic layer to traverse and reorder data structures. Using strided DMA units, gather/scatter hardware, and in-memory scratchpad buffers, the programmable near memory data rearrangement engines perform *fill* and *drain* operations to gather (resp. scatter) blocks of application data structures. In contrast to processing in memory proposals (e.g. [12]), our goal is to accelerate data access, making it possible for the many CPU cores to compute on complex data structures efficiently packed into cache.

Our system is designed to benefit data intensive applications with access patterns that have little spatial or temporal

[*] Work done as student intern at LLNL.

MEMSYS '15, October 05-08, 2015, Washington DC, DC, USA
© 2015 ACM. ISBN 978-1-4503-3604-8/15/10…$15.00
DOI: http://dx.doi.org/10.1145/2818950.2818986

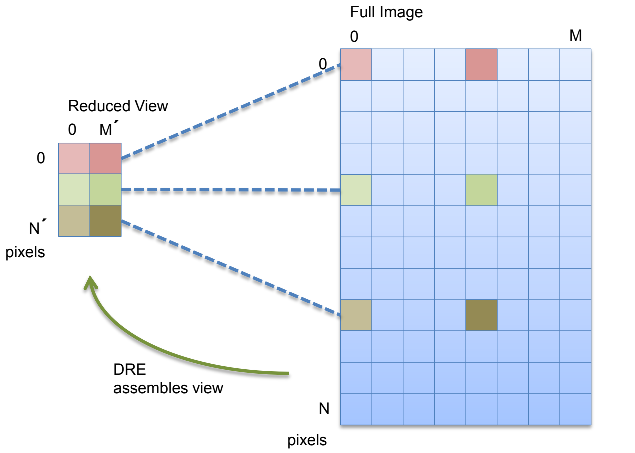

Figure 1: Full and reduced resolution images. Ideally the CPU would access the reduced view rather than the full image.

access locality. Such access patterns occur even in strided streaming accesses when the stride length exceeds cache line size. Figure 1 shows an example from image processing: extracting a reduced resolution image from full resolution. The application accesses every fourth pixel, resulting in wasted memory bandwidth and cache occupancy for the unused portions. Ideally the reduced resolution image on the left would be accessed by the CPU. Other examples include switching between row-wise and column-wise access to arrays, sparse matrix operations [4], and pointer traversal.

In this work we explore the design of a data rearrangement engine (DRE) and its interaction with the CPU, with emphasis on the API. Within the DRE microarchitecture, we focus on alternative implementations of an in-memory scratchpad buffer used to hold rearranged data. Specifically, we compare the relative merits of an SRAM vs. DRAM scratchpad in performance and energy.

2. DATA REARRANGEMENT ARCHITECTURE

Our proposed architecture targets data-centric applications for which memory latency is the primary bottleneck, resulting in frequent stalls while cache lines are evicted and filled. For these applications, data access might involve multiple loads from dispersed locations in which a small fraction of a cache line fetched from memory is actually used. DMA and gather/scatter hardware integrated with the CPU allows the CPU to initiate data structure rearrangement, but the full data structure must still traverse the memory bus. In our near-memory approach, the data rearrangement engines (DREs) are on the memory package, and only the restructured data traverses the memory bus.

2.1 DRE internals

As illustrated in Figure 2, we assume a system with CPU connected to a memory package over one or more links as in the HMC. The links connect to a switch and memory controller in the logic layer. We add the DREs into the logic layer, also connecting to the switch. Each data rearrangement engine holds a control processor (CP) and a Data Mover (DM). The CP receives commands from the CPU and in turn issues commands to a DM capable of strided and gather/scatter memory access. Both the DRE and CPU can read or write a shared memory buffer, which is used to hold a reordered view of the data.

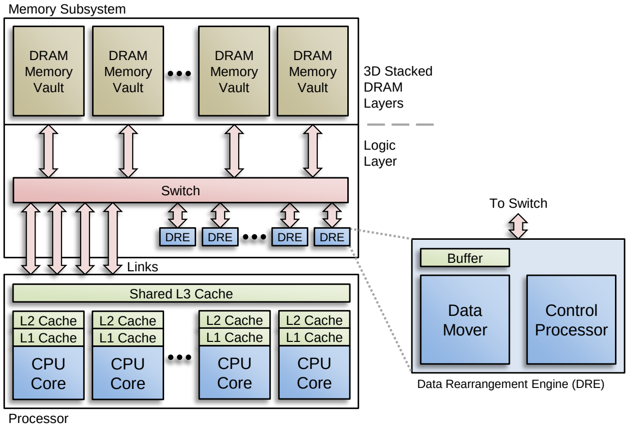

Figure 2: Near memory data rearrangement engines reside in the logic layer of 3D memory.

A DRE works as an accelerator on behalf of a requesting CPU process. The CPU part of the application sends commands to a program running on the DRE control processor. The CP sends commands to the DMA engine to transfer data between DRAM banks and the view buffer. Both CPU and DRE must explicitly manage cache coherence. Since the DRE is issuing memory requests independently of the CPU, virtual-to-physical address translation must also be handled.

2.2 API

Upon request, an application process on the CPU acquires a DRE. The application process specifies a CP program, and the Operating System loads the program into CP instruction memory, which is also in the logic layer. The main application and CP program communicate through a memory mapped address range: the application issues commands and receives completion notification by writing and reading pre-defined addresses. Commands include:

setup to load parameters, such as base addresses and either DMA size and stride for DMA operations, or index vector size and base address for gather/scatter;

fill to copy from DRAM to the view buffer according to the access pattern established during setup;

drain to copy from the view buffer into DRAM according to the access pattern established during setup.

For example in the case of the reduced resolution image, the application issues **setup** to initialize the base addresses for both full resolution and reduced resolution images, the transfer block size, and the desired stride in a specific topology (e.g. 2D or 3D image). The application then iteratively executes the following steps: issues the **fill** command to tell the DRE to fill the view buffer with a reduced resolution image block; computes the desired operations (CPU instructions) on each block; and issues **drain** to store the block to DRAM. If the original data is only read and not modified, a call to **drain** is not needed. On the DRE side, the CP waits for a command, runs the code for that command, and sends a response message back to the application when the command has completed. To execute a **setup** command, the CP copies parameters into internal registers. To execute a **fill** or **drain** command, the CP sends a stream of fine-grained operations to the Data Mover which accesses memory in a contiguous, strided or indexed access pattern.

2.3 Synchronization

The CPU and DRE exchange control messages so that the CPU can issue commands and await completion, and similarly the DRE waits for commands, executes them, and notifies completion. The interface is a set of reserved memory addresses. A set of addresses is reserved for the CPU to write a command and its associated parameters. The DRE receives a command message and sets internal state in the CP according to the specified parameters. The CPU polls for a completion message, which is sent by the DRE when the command is completed. While the present implementation uses polling, lightweight event notification mechanisms could be used as more efficient alternatives. Command responses can be decoupled from requests to allow for more parallelism between units.

2.4 Consistency

The CPU accesses physical memory through a cache hierarchy. Depending on the implementation, the DRE may also use a cache. With multiple coherence domains, the problem of consistency arises. This problem is present whenever multiple processors in disjoint coherence domains access shared data. In our approach, the application's CPU and DRE components cooperate to maintain DRAM consistency by issuing cache flush and invalidate operations at well defined synchronization points such as preceding and following fill operations. The flush/invalidate is done to the entire cache or to an address range, depending on the size of the updated region. Our implementations select the most efficient option. The overhead of maintaining consistency must be factored into evaluating the potential benefit of DREs.

When communicating between the CPU and the DRE through shared memory, the sender must flush any generated data from cache and the receiver must invalidate any cache lines associated with the memory region. Cache management on memory regions is often implemented with CPU instructions that flush or invalidate a single cache line associated with a given virtual address. Flushing or invalidating a block of memory requires repeated execution of these cache instructions for each cache-line-sized segment in the block. Furthermore, each cache level typically has its own set of cache management instructions. An optimization used in our approach is to only enable L1 cache for scratchpad memory, which is used for the view buffer and possibly a few other select structures shared by the CPU and DRE. Disabling L2 and higher caches for scratchpad memory can reduce the number of cache management operations by a factor of two or more. Since access patterns within scratchpad memory are often streaming in nature, higher level caches provide little or no benefit for this region.

2.5 Address translation

The CPU's Memory Management Unit (MMU) translates process virtual addresses to physical memory addresses. Memory requests from the DRE must also be translated, which requires that the DRE have its own address translation table. A general mechanism of mirroring the CPU MMU is done in graphics processors and high performance networking such as Infiniband. A similar mechanism has been designed for some processing-in-memory approaches, e.g. [12]. We propose a simpler, but more restrictive approach and require that data being accessed by the DRE resides in contiguous physical pages. This can be accomplished by using a custom allocator to gather a large contiguous physical range, as is being developed in transparent large page support in the OS, and by pinning pages (commonly done by network interfaces in High Performance Computing systems) to prevent subsequent relocation. The setup command gives the base physical page address, and the DRE adds the base to each address being loaded or stored. This requires minimal additional hardware, adds no performance overhead as the address can be assembled as it is loaded onto the request queue, and virtually no energy overhead since it involves a simple concatenation of bit fields.

2.6 Emulator

We have prototyped the DRE in Programmable Logic on a Xilinx Zynq 7000 System on Chip, as shown in Figure 3. The Zynq contains hard IP (labeled Processing System) and programmable logic (PL). The PS includes two ARM cores with private L1 and shared L2 caches and an SRAM scratchpad. There is also a hard memory controller in the PS and a hard AXI interconnect. The PL can hold arbitrary FPGA logic blocks. In the emulator, we have instantiated a DRE and trace capture logic. The development board has two 1 GB DDR3 memories, labeled Program DRAM and Trace DRAM. Program Memory holds the application instructions and data, while the Trace DRAM records memory read and write requests. Traces are captured non-intrusively by monitoring traffic on the AXI interconnect.

The DRE subcomponents are shown in Figure 4. The control processor is emulated using the Xilinx MicroBlaze soft processor. The CP includes a small cache, which must be synchronized with the main CPU through explicit flush and invalidate instructions. The CP executes instructions from block RAM and receives CPU commands from the Stream Switch connection to the Host Adapter. The CP sends commands to the Data Mover through the Stream Switch as well.

In our design, the main application code runs on the ARM cores, issuing instructions to the MicroBlaze in programmable logic. The ARM, MicroBlaze, and Data Mover can all read and write the Program Memory. Each of these components uses a different clock. The ARM has several programmable clocks capable of frequencies ranging from under one MHz to hundreds of MHz. The clock frequency in the PL depends on the complexity of the hardware design, but is usually not much more than 200 MHz. The program DDR3 memory is clocked at 533 MHz resulting in 1,066 megatransfers per second. These clocks must be reconciled to emulate a reasonable processor/DRE/memory system. We program the ARM and DRE clocks to coordinated frequencies and use delay units in programmable logic to increase memory access time by a corresponding amount. By running the ARM core at a slower frequency and routing memory transactions through the delay units attached to the AXI bus, we emulate a hypothetical 2.57 GHz 32-bit ARM core with a 32-byte cache line and 5 GB/s memory

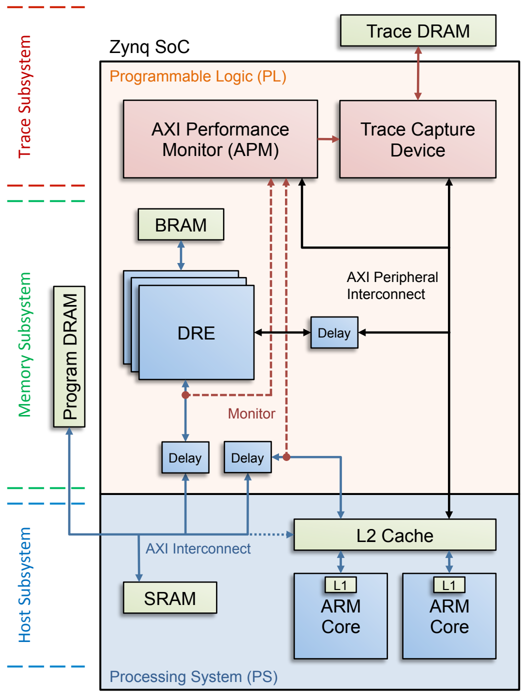

Figure 3: Zynq SoC with emulation framework

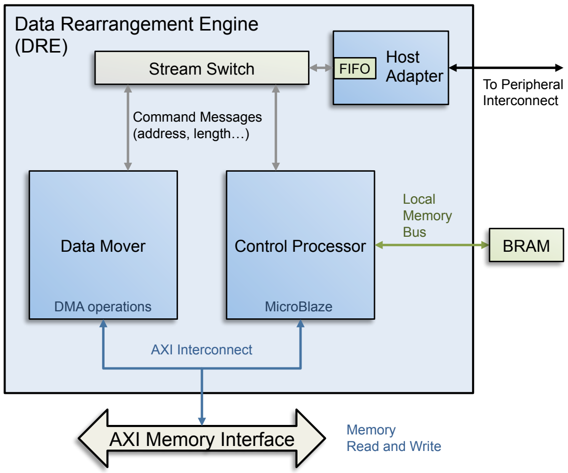

Figure 4: Data rearrangement engine consists of a data mover and control processor

bandwidth. The CP is emulated at 1.25 GHz with a 32-bit data path, 16-byte cache line, and 5 GB/s of memory bandwidth. The Data Mover runs at 1.25 GHz with a 64-bit internal data path and a bandwidth of 10 GB/s.

DRAM delay is modeled at 45 ns, and SRAM at 10 ns. We also model delay in the internal memory request queues at 20 ns and the round-trip latency on the serial link between memory package and CPU at 24 ns. The DRAM, queue, and link delays were derived from measurement on an Arira Design HMC board [2], and the SRAM latency is an estimate from the literature.

We use a simple energy model of 19.4 pJ/bit for DRAM, 1.0 pJ/bit for SRAM, and 10.3 pJ/bit for off-chip traversal. The DRAM and off-chip estimates were derived from measurement on the Arira Design board, and the SRAM estimate is from the literature. In the emulation environment, the application program runs in standalone mode, with one-to-one virtual-to-physical mapping.

3. BENCHMARKS

To quantitatively evaluate the potential benefits of in-memory data rearrangement for better performance and energy savings, we have implemented several benchmark programs. The benchmarks were chosen because they were observed not to benefit from cache hierarchy, having little cache line reuse and a high degree of partial cache line access. The benchmarks were rewritten using the CPU-DRE API (Section 2.2). Both original and DRE versions of the benchmarks were run in the emulator. All versions of the benchmarks are serial, and in the DRE versions, there is no overlap between CPU and DRE operations.

3.1 Image Differencing

The image differencing benchmark is an adaptation of the reduced resolution image example of Figure 1. This benchmark computes the pixel-wise difference of two reduced resolution images, storing the difference image to memory. The DRE version uses the Data Mover's strided access functions to assemble the sub-sampled images. The application allocates two view buffer areas to hold a portion of each reduced resolution image and defines the decimation factor as a parameter to the setup command. The DRE uses this parameter to set the stride for DMA operations. In each iteration of a loop, the CPU issues the fill command to the DRE. The DRE fills the view buffers with pixels sub-sampled from the original images, and notifies the CPU of completion. The CPU then invalidates its cache, which will then be filled from the updated view buffers, and takes the pixel difference, writing out the reduced size output image block to memory. In the evaluation, a 16X decimation factor was used.

3.2 RandomAccess

This benchmark [1], also known as "GUPS" (for giga updates per second), is representative of extremely irregular applications and is designed to measure memory performance in the presence of a completely random access pattern. The benchmark reads, modifies and writes back random elements of a table that occupies up to half the total memory size. In our benchmark the table is of size .5 GB. The benchmark iteratively performs the computation shown in Figure 5 where `ran` is a sequence of random 64-bit numbers.

The DRE version uses the Data Mover's gather/scatter

```
T[ran[j] & (TableSize-1)] ^= ran[j];
```

Figure 5: RandomAccess xor calculation updates a random location.

functions to collect 64-bit elements from T indexed by the ran vector. The CPU part of the application then performs an XOR on the gathered elements, and then uses the DRE to scatter the updated table elements. A detailed control flow is shown in Figure 6. Before entering the depicted flow, caches lines are assumed to be in a clean or invalid state. First, the table T is initialized and explicitly flushed to memory. Then the setup procedure is called to establish a reordered view of the table elements based on the index array. After the view is setup, the main loop repeatedly processes blocks of array elements by filling, computing on, and draining the view buffer. The index array is initialized with new values and flushed before each block is processed. The CPU also sends a message to the CP to invalidate the index array. Since the DM does not have a cache, no cache management operations are required for this processing unit. Cache maintenance is also performed on the view buffer after the computation step so that the view is consistent for the next iteration. The fill and drain procedures start by sending a message to the CP which then sequences the work done by the DM. After configuring the index access pattern in the DM, the CP then sends a stream of indexes to the DM to perform the gather or scatter operation. Even though a synchronous control flow is shown between the processing units, responses can be decoupled from requests to allow for overlapped operations and more parallelism.

3.3 PageRank

PageRank is a well-known data intensive algorithm that ranks web pages by their relative importance. PageRank models a web surfer that randomly follows links with random restart. It is often iteratively computed as a stochastic random walk with restart, where the initial page rank vector is a uniformly distributed random number across all vertices of a web graph. In this benchmark, we generate a synthetic graph using the RMAT generator from the Graph500 benchmark with edge factor 16 (edge factor is defined as the ratio of the graph's edge count to its vertex count). We use an adjacency list representation of the graph in which a list element holds the vertex id and its edge list, i.e. the ids of its edge targets. The page rank vector contains one 64-bit floating point number for each vertex. The algorithm iterates through the list of vertices and updates each vertex's rank based on its connections.

In the DRE-assisted version, the setup and fill commands are repeated for each vertex with a minimum number of edges. The fill command uses a vertex's edge list to gather neighboring page rank values into the view buffer. The CPU then calculates the rank from the values in the view buffer and writes back a single value without the need for the drain command. Since the edge lists are implemented as separate index arrays in memory, the DRE setup command must be called to establish a different view in each instance. In the benchmark evaluation, a 2^{22} vertex scale free graph was used.

3.4 Sparse Matrix to Vector Multiply

This benchmark multiplies a sparse matrix with a dense vector. Its access pattern is similar to PageRank in that array elements are indexed out of order. Likewise, only the gather function is needed since the source matrix and vector are read only with the result going to another dense vector.

The benchmark code is adapted from Berkeley's SpMV BeBOP implementation [7], which creates banded synthetic matrices similar to those used in real world applications. Typical matrices range in size from dozens to millions of rows and columns with between 5 and 200 non-zero elements per row. The banded structure of the matrix is created by placing a large percentage of the non-zero elements in the primary diagonal section.

The sparse matrices are stored in Compressed Row Storage (CRS) format, requiring an array to hold all non-zero elements in the matrix, an array to hold the column index of each element, and an array to hold the row pointers. These data structures are sequentially accessed and can take advantage of pre-fetch hardware in conventional systems. The dense vector is indirectly accessed by the gather hardware through the column index array.

The CPU part of the applications does a setup by passing the DRE a pointer to the dense vector, its element size, and a pointer to the column index array. On each iteration the CPU sends the DRE a fill command: it passes the DRE the address and size of the output block along with its offset in the view (for use when the block size is less than the view size). The output block serves as a window into the view established with the setup command. The CP issues instructions to the DM to gather array elements into the view buffer according to the access pattern specified in the setup command. When the fill command is completed, the CPU invalidates its cache so that it gets the updated view buffer. Each value from the matrix source vector is then multiplied with the corresponding value in the block and accumulated. The accumulated value is stored by the CPU into the result vector.

The major modification in the SpMV application comes from setting up the data efficiently to use the view buffer. A straightforward method would simply fill the view buffer with a row of data, let the DRE rearrange that data, and perform the computation on the output block. However, for sparse matrices, where many rows contain only dozens (or fewer) of non-zero elements, the overhead associated with filling the view buffer is greater than the memory latency to access the elements directly. To achieve performance gain, the rows are batched together to fill the view buffer. In the benchmark, the sparse matrix is of size 2M×2M with 34 non-zero elements per row.

4. EVALUATION

Using the emulator configured to model the latency of a Gen2 HMC, we evaluated the DRE on the benchmark set. Metrics consist of performance as measured by elapsed (emulated) time and energy. Parameters of the study include view buffer type and memory access unit. The view buffer scratchpad memory is an integral part of our DRE system. It is used by both CPU and DRE to exchange data blocks. It enables a batched style of interaction, so that each processor can hand off an entire buffer of data rather than single data items. The latency of synchronization between CPU and CP makes it desirable to give the DRE large blocks of data rearrangement tasks, and the view buffer makes this

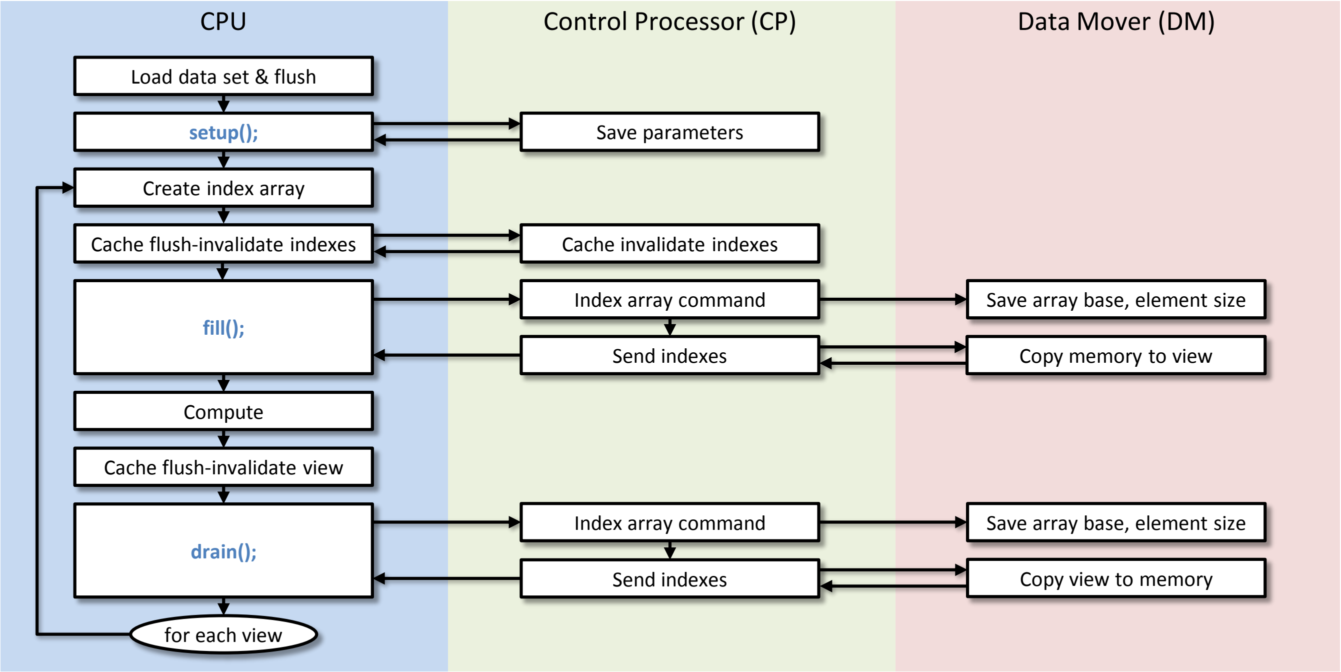

Figure 6: Control flow for gather-scatter operations on an array showing usage of setup, fill and drain procedures.

possible.

Reserving a portion of the DRAM for the view buffer is the simplest from a hardware design point of view – no changes are required to the logic layer or the DRAM layers – and also affords the most flexibility, since each application can size the view buffer to its requirements. It does slightly reduce DRAM capacity available to the application.

Alternatively, SRAM scratchpads can be sited on the logic layer to be used as a view buffer and as a general purpose scratchpad memory for the control processor. SRAM is significantly faster than DRAM and incurs a lower energy cost to access small data items. However, SRAM introduces additional complexity into the logic layer at a higher cost, and it is smaller in capacity than DRAM. In this study we compare the benchmarks' performance and energy with a DRAM or an SRAM view buffer.

The second parameter is minimum access unit. The Gen2 HMC has a minimum size of 32 bytes. Even when a 16-byte packet is requested, a full 32 bytes are accessed. The irregular access patterns of our benchmark set often read or write 8-byte data items. Therefore, we model a hypothetical "narrow" memory capable of 8-byte accesses.

A previous study using the SRAM view buffer [10] additionally considered link latency at three settings, none (0 ns), average (20 ns) and high (40 ns), to better model usage patterns ranging from lightly to heavily loaded memory subsystem. In this work, we report results for the average case of 20 ns.

The performance of the SRAM and DRAM view buffer designs are compared in Figures 7. Speedup of the DRE versions is normalized to the CPU only version. All the benchmarks show speedup over the CPU only, but using an SRAM view buffer gives consistently higher speedup. With a DRAM view buffer, speedup is between 61% – 93% of SRAM.

Energy reduction is shown for full memory access mode

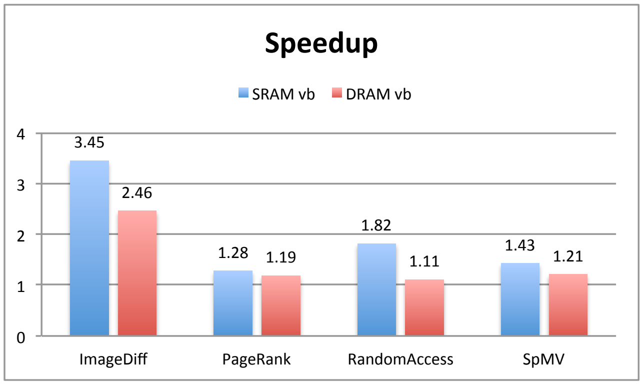

Figure 7: Speedup of CPU+DRE vs. CPU only. vb: view buffer

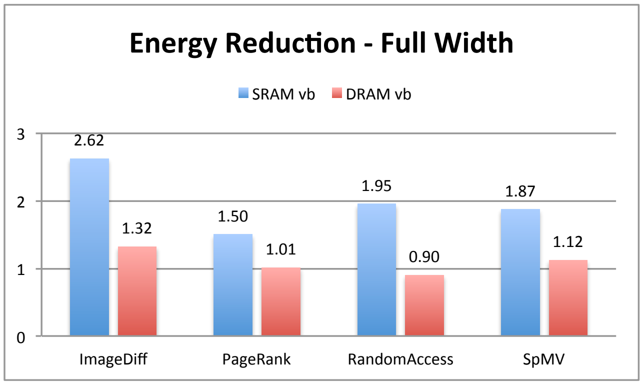

Figure 8: Energy reduction for full width memory access of CPU+DRE vs. CPU only. vb: view buffer

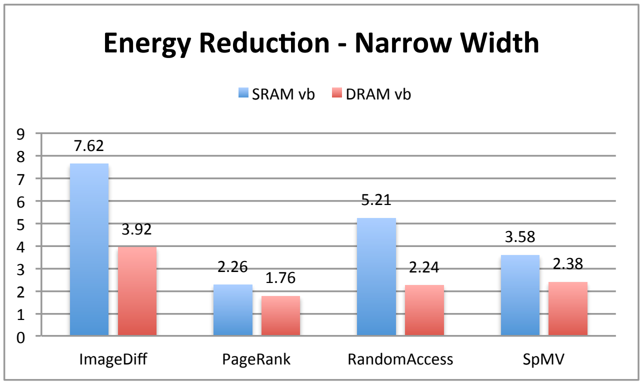

Figure 9: Energy reduction for narrow width memory access of CPU+DRE vs. CPU only. vb: view buffer

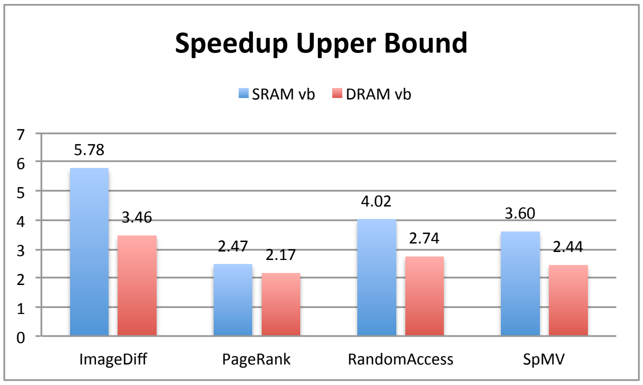

Figure 10: Speedup relative to maximum achievable. vb: view buffer

with widths of 32 bytes and narrow access mode with widths of 8 bytes in Figures 8 and 9 respectively. Energy is reduced in all cases with SRAM, but only for ImageDiff and SpMV with a DRAM view buffer. In fact, RandomAccess uses more energy when constrained to full-width DRAM accesses. Narrow access mode shows energy savings in all benchmarks for both SRAM and DRAM view buffer designs. Results indicate an advantage to the SRAM view buffer design and to the narrow memory access mode. The DRAM design shows a benefit in both speedup and energy in narrow mode, and speedup in full access mode.

Figure 10 shows measured CPU+DRE speedup relative to the maximum achievable. The upper bound is calculated by setting the DRE time to zero, that is, either the DRE is infinitely fast or the DRE time is completely hidden by overlapped computation on the CPU. Our study measured sequential execution of a single CPU core and a single DRE in which they each wait for the other: the CPU gives the DRE a command and then waits for the command to complete before going to the next stage of work, and the DRE waits for a command, executes it, and then goes idle until it gets another command.

Comparing the upper bounds of SRAM and DRAM view buffers, it is evident that the speedup potential is higher for SRAM view buffers. Even when the effective DRE time is zero, the slower DRAM access time for the CPU to read and write the view buffer limits the maximum achievable speedup.

5. CONCLUSIONS AND FUTURE WORK

In this work we have designed a novel off-load engine suitable for placement in a logic layer of emerging 3D memory stacks. The Data Rearrangement Engine performs in memory restructuring of scattered data items into compact view buffers that are accessed by applications running in the CPU. The view buffer contains only data that is needed by the CPU, assuring high cache occupancy. Further, computation on the data can take advantage of vector and SIMD units in the CPU, which would not be possible in the original data structure. In our benchmarks, the DRE-assisted ImageDiff and RandomAccess benchmarks could be vectorized on the emulator's ARM NEON processor through compiler directives. However, the double-precision, floating-point operations in PageRank and SpMV were not vectorized because the NEON processor used on the emulator does not support double-precision operations. In future work, we will use a newer ARM-based emulator with double-precision SIMD support to measure performance on vectorized floating-point operations.

Our study was done with serial, non-overlapped benchmarks. In the future, we plan to overlap computation and communication in the single CPU single DRE configuration by using multiple view buffers. It will likely be possible to achieve even more speedup with multiple cooperating threads using their own CPU/DRE instances up to the point of saturating memory bandwidth. Comparing achieved speedup in the single core single DRE serial mode to the upper bound shows that there is room for further speedup once these common optimizations have been applied.

Finally, we will work with a larger application set to evaluate the choice of address translation methods. While a simple base and bounds mechanism is more efficient, it may not scale to terabyte size data sets without considerable application refactoring, if at all. We will evaluate the use of more general address translation tables, in particular, the overhead of keeping the tables synchronized and the added latency to load new table entries during fill or drain operations.

6. ACKNOWLEDGMENTS

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under contract No. DE-AC52-07NA27344. This work was supported by Lawrence Livermore National Laboratory LDRD project 013-ERD-25 Data-centric Architectures. LLNL-PROC-675466

7. REFERENCES

[1] RandomAccess. http://icl.cs.utk.edu/projectsfiles/hpcc/RandomAccess/, 2012.

[2] Arira design. http://www.ariradesign.com, 2015.

[3] J. B. Brockman, S. Thoziyoor, S. K. Kuntz, and P. M. Kogge. A low cost, multithreaded processing-in-memory system. In *Workshop on Memory Performance Issues at ISCA*, pages 16–22. ACM, 2004.

[4] P. C. Diniz and J. Park. Data reorganization engines for the next generation of system-on-a-chip fpgas. In *Proceedings of the 2002 ACM/SIGDA Tenth*

International Symposium on Field-Programmable Gate Arrays, FPGA '02, pages 237–244, New York, NY, USA, 2002. ACM.

[5] J. Draper, J. T. Barrett, J. Sondeen, S. Mediratta, C. W. Kang, I. Kim, and G. Daglikoca. A prototype processing-in-memory (PIM) chip for the data-intensive architecture (diva) system. *The Journal of VLSI Signal Processing*, 40:73–84, 2005.

[6] B. B. Fraguela, J. Renau, P. Feautrier, D. Padua, and J. Torrellas. Programming the FlexRAM parallel intelligent memory system. In *Proceedings of the Ninth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '03, pages 49–60, New York, NY, USA, 2003. ACM.

[7] H. Gahvari, M. Hoemmen, J. Demmel, and K. Yelick. Benchmarking sparse matrix-vector multiply in five minutes. In *SPEC Benchmark Workshop*, Austin, TX, January 2007. http://bebop.cs.berkeley.edu.

[8] J. Gebis, S. Williams, C. Kozyrakis, and D. Patterson. VIRAM1: A media-oriented vector processor with embedded DRAM. In *Student Design Contest, DAC*, 2004.

[9] M. Gokhale, W. Holmes, and K. Iobst. Processing in memory: The Terasys massively parallel PIM array. *IEEE Computer*, 28(4):23–31, Apr 1995.

[10] S. Lloyd and M. Gokhale. In-memory data rearrangement for irregular, data intensive computing. In *IEEE Computer special issue on Irregular Applications*, Aug. 2015.

[11] Micron. Hybrid Memory Cube. http://www.hybridmemorycube.org/, 2011.

[12] R. Nair, S. Antao, C. Bertolli, P. Bose, et al. Active Memory Cube: A processing-in-memory architecture for exascale systems. *IBM Journal of Research and Development*, 59(2/3):17:1–17:14, March-May 2015.

[13] Y. Solihin, J. Lee, and J. Torrellas. Using a user-level memory thread for correlation prefetching. In *Proceedings of the 29th annual international symposium on Computer architecture*, ISCA '02, pages 171–182, Washington, DC, USA, 2002. IEEE Computer Society.

[14] H. S. Stone. A logic-in-memory computer. *Computers, IEEE Transactions on*, C-19(1):73–78, Jan 1970.